LA-UR-18-25404
Approved for public release; distribution is unlimited.

Title: Debugging Computer Code

Author(s): Budge, Kent Grimmett

Intended for: Parallel Computing Summer School Talk

Issued: 2018-06-20

Debugging Computer Code

Kent G. Budge
CCS-2, Computational Physics and Methods

Introduction

Debugging? Klingons do not debug. Our software does not coddle the weak.

For the rest of us, finding and removing defects in code is a fundamental programming skill. Besides preventing code from performing its intended function, bugs destroy users' confidence in the reliability of code even when no errors are obvious.

A through discussion of debugging techniques could easily fill a semester course in computer science. We have an hour here today.

I will take that time to discuss broad principles rather than fine points of debugging.

Ancient History

Python? That is for children. A Klingon Warrior uses only machine code, keyed in on the front panel switches in raw binary.

When I first began programming, in 1976, on a PDP 11/40, there was no such thing as a symbolic debugger. “Debug by printf” was not the last resort; it was the *only* resort, short of repeated code inspection.

Fortunately, this machine had 128K of core memory, no mechanism for page swapping, and a disk the size of a washing machine that held 2 Mbyte of storage.

There simply wasn’t room to cram very many bugs onto this machine.

(But, *man*, did it have a beautifully orthogonal instruction set!)

How to Deliver Bug-Free Code

Perhaps it IS a good day to die! I say we ship it!

- Don’t write buggy code.
- Scaffold code during development.
- Test comprehensively.
- Be proficient in using a good symbolic debugger and heap inspector.
- Close the loop.

Don’t write buggy code

Specifications are for the weak and timid!

- Know what you want your code to do before you write it.
- Compartmentalize, compartmentalize, compartmentalize
  - Globals are evil
  - Keep functions short and simple
  - Program with objects
  - Use enumerations and constants rather than magic numbers
- Don’t reinvent the wheel
  - Use standard libraries (STL)
  - Use *reliable* third-party libraries
    - How do you know it’s reliable?
- Avoid inherently buggy programming techniques
  - Case instead of long succession of ifs
  - Polymorphism instead of case
  - Avoid raw pointers and arrays

Scaffold code during development

- Use modern configuration tools like cmake to switch between debug and production builds.
- Debug builds should be liberally sprinkled with checks that can disappear in production builds.
  - NDEBUG
- Design by Contract:
  - Preconditions
  - Postconditions
  - Invariants
  - Miscellaneous checks
- Production code debugging options
  - “Pay to play”

Design by Contract

```
template <class RandomContainer>
void rotate(RandomContainer &r, RandomContainer &qt, const unsigned n,
            unsigned i, double a, double b)
{
  Require(r.size() == n * n);
  Require(qt.size() == n * n);
  Require(i + 1 < n);

  // … the good stuff ...

  Ensure(r.size() == n * n);
  Ensure(qt.size() == n * n);
}
```

Test comprehensively

You question the worthiness of my code? I should kill you where you stand!

- Use automated unit testing
  - Levelized code
- Use a code coverage tool
  - There is almost no point of diminishing return with code coverage. A code that is 95% covered is a code that is inadequately tested.
  - Coverage by function is a good start. Coverage by branch *must* follow.

Actual debugging

- Use a good symbolic debugger.
  - Should display objects sensibly.
  - Should allow variables to be set by hand in the middle of a session.
  - Should make it easy to pull up the suspect function.
  - Should allow conditional breakpoints.
- Use a good memory checker.
  - Should detect all out-of-bounds errors.
  - All else is frosting.

Isolating bugs

- Think a little before jumping into the debugger.
- Strip input to bare minimum that reproduces the bug.
- Previous versus head build.
- Beware bug that is actually a feature.
- Restart

When all else fails...

- Fire up the debugger on the minimum failing input.
- Identify a “bug trace” – output or variable value that betrays the error.
- Close in by bisection.
- Once within a manageable range of code, begin stepping through code with the bug trace monitored.
- Peel the onion.

Close the loop

By filing this bug report, you have questioned the honor of my family. Prepare to die!

- “With many eyes, all bugs are shallow”
  - Know your customers and talk to them.
  - The burden is on you, not them, to isolate the bug.
  - No one ever reads the manual.
- Consider open-sourcing your code
  - In the LANL environment, there are obvious restrictions on open sourcing, but it can be done in appropriate cases.
- Every bug report should add at least one new test case to your test suite.